DZone Refcardz

CONTENTS INCLUDE:

Core .NET

By Jon Skeet

ABOUT .NET

The .NET Framework has been growing steadily since its birth—the API for .NET 3.5 is far bigger than that of .NET 1.0. With so much to remember, you'll find this refcard useful for those core pieces of information which you need so often but which (if you're like me) you can never quite recall without looking them up—topics like string formatting, and how to work with dates and times effectively. This reference card deals only with the core of .NET, making it applicable for whatever kind of project you're working on.

COMMON .NET TYPES

The .NET Framework has a massive set of types in it, but some are so important that C# and VB have built-in keywords for them, as listed in table 1.

C# Alias	VB Keyword	.NET Type	Size(bytes)
object	Object	System.Object	12 (8 bytes are normal overhead for all reference types)
string	String	System.String	Approx. 20 + 2*(length in characters)
bool	Boolean	System.Boolean	1
byte	Byte	System.Byte	1
sbyte	SByte	System.SByte	1
short	Short	System.Int16	2
ushort	UShort	System.UInt16	2
int	Integer	System.Int32	4
uint	UInteger	System.UInt32	4
long	Long	System.Int64	8
ulong	ULong	System.UInt64	8
float	Single	System.Single	4 (accurate to 7 significant digits)
double	Double	System.Double	8 (accurate to 15 significant digits)
decimal	Decimal	System.Decimal	16 (accurate to 28 significant digits)
char	Char	System.Char	2
n/a	Date	System.DateTime	8

Table 1. Common types and their language-specific aliases

Apart from Object and String, all the types above are value types. When choosing between the three floating point types (Single, Double and Decimal):

- For financial calculations (i.e. when dealing with money), use Decimal
- For scientific calculations (i.e. when dealing with physical quantities with theoretically infinite precision, such as weights), use Single or Double

The Decimal type is better suited for quantities which occur in absolutely accurate amounts which can be expressed as decimals: 0.1, for example, can be expressed exactly as a decimal but not as a double. For more information, read http://pobox.com/~skeet/csharp/decimal.html and http://pobox.com/~skeet/csharp/floatingpoint.html.

FORMATTING STRINGS

One common task which always has me reaching for MSDN is working out how to format numbers, dates and times as strings. There are two ways of formatting in .NET: you can call ToString directly on the item you wish to format, passing in just a format string or you can use composite formatting with a call to String.Format to format more than one item at a time, or mix data and other text. In either case you can usually specify an IFormatProvider (such as CultureInfo) to help with internationalization. Many other methods in the .NET Framework also work with composite format strings, such as Console.WriteLine and StringBuilder.AppendFormat.

Composite format strings consist of normal text and format items containing an *index* and optionally an *alignment* and a *format string*. Figure 1 shows a sample of using composite format string, with each element labeled.

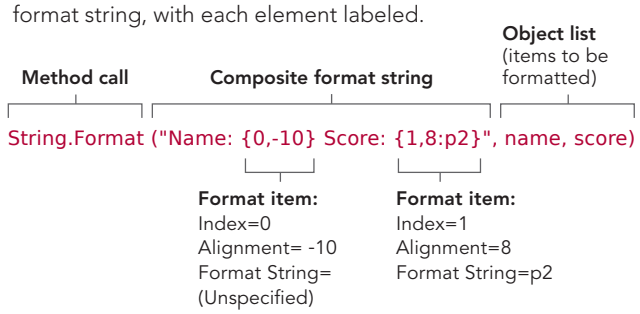


Figure 1. The anatomy of a call to String.Format

When the alignment isn't specified you omit the comma; when the format string isn't specified you omit the colon. Every format item must have an index as this says which of the following arguments to format. Arguments can be used any number of times, and in any order. In general, the alignment is used to →

Formatting Strings, continued

specify a minimum width – if this is negative, the result is padded with spaces to the right; if it's positive, the result is padded with spaces to the left. The effect of the format string depends on the type of item being formatted. To include a brace as normal text in a composite format string (instead of it indicating the beginning or end of a format item), just double it. For example, the result of String.Format("{{0}}") is "{0}".

Numeric format strings

Numbers can be formatted in two ways: with *standard* or *custom* format strings. The standard ones allow some flexibility in terms of the precision and style, but the custom ones can be used for very specific formats.

Standard numeric format strings

Standard numeric format strings all take the form of a single letter (the *format specifier*) and then optionally one or two digits (the *precision*). For example, a format string of N5 has N as the format specifier and 5 as the precision. The exact meaning of the precision depends on the format specifier, as described in table 2. If no precision is specified a suitable default is used based on the current IFormatProvider.

Format	Description	Precision	Examples (with US English IFormatProvider)
C or c	Currency – exact format is specified by NumberFormatInfo	Number of decimal places	123.4567, "c" => "$123.46" 123.4567, "c3" => "$123.457"
D or d	Decimal (integer types only)	Minimum number of digits	123, "d5" => "00123" 123, "d2" => "123"
E or e	Scientific – used to express very large or small numbers in exponential format.	Number of digits after the decimal point	123456, "e2" => "1.23e+005" 123456, "E4" => "1.2345E+005"
F or f	Fixed point	Number of decimal places	123.456, "f2" => "123.46" 123.4, "f3" => "123.400"
G or g	General—chooses fixed or scientific notation based on number type and precision	Depends on exact format used (see http://msdn.microsoft.com/en-us/library/dwhawy9k.aspx for details)	123.4, "g2" => "1.2e+02" 123.4, "g6" => "123.4" 123.400m, "g" => "123.400"
N or n	Number—decimal form including thousands indicators (e.g. commas)	Number of decimal places	1234.567, "n2" => "1,234.57"
P or p	Percentage—number is multiplied by 100 and percentage sign is applied	Number of decimal places	0.1234, "p1" => "12.3 %"
R or r	Round-trip—if you later parse the result, you're guaranteed to get the original number.	Ignored	0.12345, "r" => 0.12345
X or x	Hexadecimal (integer types only). The case of the result is the same as the case of the format specifier.	Minimum number of digits	123, "x" => "7b" 123, "X4" => "007B"

Table 2. Standard numeric format strings

Custom numeric format strings

To format numbers in a custom fashion, you provide a pattern to the formatter, consisting of format specifiers as shown in table 3.

Custom numeric format strings, continued

Format Specifier	Name	Description
0	Zero placeholder	Always formatted as 0 or a digit from the original number
#	Decimal placeholder	Formatted as a digit when it's a significant digit in the number, or omitted otherwise
.	Decimal point	Formatted as the decimal point for the current IFormatProvider
,	Thousands separator and number scaling specifier	When used between digit or zero placeholders, formatted as the group separator for the current IFormatProvider. When it's used directly before a decimal point (or an implicit decimal point) each comma effectively means "divide by a thousand".
%	Percentage placeholder	Formatted as the percent symbol for the current IFormatProvider, and also multiplies the number by 100
‰ (\u2030)	Per mille placeholder	Similar to the percentage placeholder, but the number is multiplied by 1000 instead of 100, and the per mille symbol for the culture is used instead of the percent symbol.
E0, e0, E+0, e+0, E-0, or e-0	Scientific notation	Formats the number with scientific (exponential) notation. The number of 0s indicates the minimum number of digits to use when expressing the exponent. For E+0 and e+0, the exponent's sign is always expressed; otherwise it's only expressed for negative exponents.
" or '	Quoting for literals	Text between quotes is formatted exactly as it appears in the format string (i.e. it's not interpreted as a format pattern)
;	Section separator	A format string can consist of up to three sections, separated by semi-colons. If only a single section is present, it is used for all numbers. If two sections are present, the first is used for positive numbers and zero; the second is used for negative numbers. If three sections are present, they are used for positive, negative and zero numbers respectively.
\c	Single-character escape	Escapes a single character, i.e. the character c is displayed verbatim

Table 3. Custom numeric format specifiers

Table 4 shows examples of custom numeric format strings, when formatted with a US English format provider.

Number	Format String	Output	Notes
123	####.00#	123.00	0 forces a digit; # doesn't
12345.6789	####.00#	12345.679	Value is rounded to 3 decimal places
1234	0,0.#	1,234	Decimal point is omitted when not required
1234	0,.####	1.234	Value has been divided by 1000
0.35	0.00%	35.00%	Value has been multiplied by 100
0.0234	0.0\u2030	23.4‰	
0.1234	0.00E0	1.23E-1	Exponent specified with single digit
1234	0.00e00	1.23e03	Exponent is specified with two digits, but sign is omitted
1234	##'text0'###	1text0234	The text0 part is not parsed as a format pattern
12.34	0.0;000.00; 'zero'	12.3	First section is used
-12.34	0.0;000.00; 'zero'	012.34	Second section is used
0	0.0;000.00; 'zero'	zero	Third section is used

Table 4. Sample custom numeric format strings and their results

Date and time format strings

Dates and times tend to have more cultural sensitivity than numbers—the ordering of years, months and days in dates varies between cultures, as do the names of months and so forth. As with numbers, .NET allows both standard and custom format strings for dates and times.

Standard date and time format strings

Standard date and time format strings are always a single character. Any format string which is longer than that (including whitespace) is interpreted as a custom format string. The round-trip (o or O), RFC1123 (r or R), sortable (s) and universal sortable (u) format specifiers are culturally invariant—in other words, they will produce the same output whichever IFormatProvider is used. Table 5 lists all of the standard date and time format specifiers.

Format Specifier	Description	Example (US English)
d	Short date pattern	5/30/2008
D	Long date pattern	Friday, May 30, 2008
f	Full date/time pattern (short time)	Friday, May 30, 2008 8:40 PM
F	Full date/time pattern (long time)	Friday, May 30, 2008 8:40:36 PM
g	General date/time pattern (short time)	5/30/2008 8:40 PM
G	General date/time pattern (long time)	5/30/2008 8:40:36 PM
M or m	Month day pattern	May 30
O or o	Round-trip pattern	2008-05-30T20:40:36.8460000+01:00
R or r	RFC1123 pattern (Assumes UTC: caller must convert.)	Fri, 30 May 2008 19:40:36 GMT
s	Sortable date pattern (ISO 8601 compliant)	2008-05-30T20:40:36
t	Short time pattern	8:40 PM
T	Long time pattern	8:40:36 PM
u	Universal sortable date pattern (Assumes UTC: caller must convert.)	2008-05-30 19:40:36Z
U	Universal full date/time pattern (Format automatically converts to UTC.)	Friday, May 30, 2008 7:40:36 PM
Y or y	Year month pattern	May, 2008

Table 5. Standard date and time format specifiers

Custom date and time format strings

As with numbers, custom date and time format strings form patterns which are used to build up the result. Many of the format specifiers act differently depending on the number of times they're repeated. For example, 'd' is used to indicate the day—for a date falling on a Friday and the 5th day of the month, "d" (in a custom format string) would produce "5", "dd" would produce "05", "ddd" would produce "Fri" and "dddd" would produce "Friday" (in US English—other cultures will vary). Table 6 shows each of the custom date and time format specifiers, describing how their meanings change depending on repetition.

Format Specifier	Meaning	Notes and variance by repetition
d, dd, ddd, dddd	Day	d 1-31 dd 01-31 ddd Abbreviated day name (e.g. Fri) dddd Full day name (e.g. Friday)
f, ff ... ffffff	Fractions of a second	f Tenths of a second ff Hundredths of a second (etc) The specified precision is always used, with insignificant zeroes included if necessary
F, FF ... FFFFFFF	Fractions of a second	Same as f ... ffffff except insignificant zeroes are omitted
g	Period or era	For example, "A.D."
h, hh	Hour in 12 hour format	h 1-12 hh 01-12
H, HH	Hour in 24 hour format	H 0-23 HH 00-23
K	Time zone offset	For example, +01:00; outputs Z for UTC values.
m, mm	Minute	m 0-59 mm 00-59
M ... MMMM	Month	M 1-12 MM 01-12 MMM Abbreviated month name (e.g. Jan) MMMM Full day name (e.g. January)
s, ss	Seconds	s 0-59 ss 00-59

Table 6. Custom date and time format specifiers

Custom date and time format strings, continued

Format Specifier	Meaning	Notes and variance by repetition
t, tt	AM/PM designator	t First character only (e.g. "A" or "P") tt Full designator (e.g. "AM" or "PM")
y ... yyyyy	Year	y 0-99 (least significant two digits are used) yy 00-99 (least significant two digits are used) yyy 000-9999 (three or four digits as necessary) yyyy 0000-9999 yyyyy 00000-99999
z ... zzz	Offset from UTC (of local operating system)	z -12 to +13, single or double digit zz -12 to +13, always double digit (e.g. +05) zzz -12:00 to +13:00, hours and minutes
:	Time separator	Culture-specific symbol used to separate hours from minutes, etc.
/	Date separator	Culture-specific symbol used to separate months from days, etc.
'	Quoting for literals	Text between two apostrophes is displayed verbatim.
%c	Single custom format specifier	Uses c as a custom format specifier; used to force a pattern to be interpreted as a custom instead of a standard format specifier.
\c	Single-character escape	Escapes a single character, i.e. the character c is displayed verbatim.

Table 6. Custom date and time format specifiers, continued

Typically only years within the range 1-9999 can be represented, but there are some exceptions due to cultural variations. See http://msdn.microsoft.com/en-us/library/8kb3ddd4.aspx for more details on this and all of the formatting topics.

Table 7 shows examples of custom date and time format strings, when formatted with a US English format provider. (The date and time in question is the same one used to demonstrate the standard format strings.)

Format String	Output	Notes
yyyy/MM/dd'T'HH:mm:ss.fff	2008/05/30T20:40:36.846	T is quoted for clarity only—T is not a format specifier, so would have been output anyway.
d MMMM yy h:mm tt	30 May 08 8:40 PM	12 hour clock, single digit used
HH:mm:sszzz	20:40:36+01:00	24 hour clock, always two digits
yyyy g	2008 A.D.	Rarely used—era is usually implicit
yyyyMMddHHmmssfff	20080530204036846	Not very readable, but easily sortable—handy for log filenames. Consider using UTC though.

Table 7. Sample custom date and time format strings and their results

WORKING WITH DATES AND TIMES

The support in .NET for dates and times has changed significantly over time. It's never simple to do this properly (particularly taking time zones and internationalization into account, along with all the normal worries about leap years and other idiosyncrasies) but the support has definitely improved. A full discussion of all the subtleties is beyond the scope of this reference card, but MSDN has an excellent page which goes into more depth: http://msdn.microsoft.com/en-us/library/bb384267.aspx.

I suggest you read that article and other resources, but use this reference card as a quick *aide mémoire*.

DateTime and TimeZone have been in the .NET Framework since version 1.0. DateTime simply stores the number of ticks since midnight on January 1st, 1 A.D.—where a tick is 100ns. This structure was improved in .NET 2.0 to allow more sensible time zone handling, but it's still not entirely satisfactory. It's useful when you don't care about time zones, but newer alternatives have been introduced. TimeZone is sadly restricted to retrieving the time zone of the local machine.

→

Working with dates and times, continued

.NET 2.0SP1 (which is part of .NET 3.0SP1 and .NET 3.5) introduced DateTimeOffset which is effectively a DateTime with an additional Offset property representing the difference between the local time and UTC. This unambiguously identifies an instant in time. However, it's not inherently aware of time zones—if you add six months, the result will have the same Offset even if the "logical" answer would be different due to daylight saving time.

.NET 3.5 introduced TimeZoneInfo which is a much more powerful class for representing time zones than TimeZone—the latter is now effectively deprecated. TimeZoneInfo allows you access to all the time zones that the system knows about, as well as creating your own. It also contains historical data (depending on your operating system) instead of assuming that every year has the same rules for any particular time zone.

Tips

- If you're using .NET 2.0SP1 or higher, you should consider DateTimeOffset to be the "default" date and time type. Some databases are easier to work with using DateTime, however.
- It is usually a good idea to use a UTC representation for as much of the time as possible, unless you really need to preserve an original time zone. Convert to local dates and times for display purposes.
- If you need to preserve the original time zone instead of just the offset at a single point in time, keep the relevant TimeZoneInfo.
- There are situations where the time zone is irrelevant, primarily when either just the date or just the time is important. Identify these situations early and make sure you don't apply time zone offsets.
- In almost all commonly used formats:

  10:00:00.000+05:00 means "the local time is 10am; in UTC it's 5am"

  10:00:00.000-05:00 means "the local time is 10am; in UTC it's 3pm"
- DateTimeOffset.Offset is positive if the local time is later than UTC, and negative if the local time is earlier than UTC. In other words, Local = UTC + Offset

In addition to the types described above, the Calendar and DateTimeFormatInfo classes in the System.Globalization namespace are important when parsing or formatting dates and times. However, their involvement is usually reasonably well hidden from developers.

TEXT ENCODINGS

Internationalization (commonly abbreviated to i18n) is another really thorny topic. Guy Smith-Ferrier's book, *.NET Internationalization* (Addison-Wesley Professional, 2006) is probably the definitive guide. However, before you even consider what resources, localized strings, and so forth, you will display to your user, you need to make sure you can accurately move textual data around—which means you need to know about encodings.

Text encodings, continued

Whenever you use a string in .NET, it uses *Unicode* for its internal representation. Unicode is a standard way of converting characters ('a', 'b', 'c', etc.) into numbers (97, 98, 99 respectively, in this case). Each number is a 16 bit unsigned integer—in other words it's in the range 0-65535.. You need to be careful how you read your text to start with, and how you output it. This almost always involves converting between the textual representation (your string) and a binary representation (plain bytes)—either in memory or to disk, or across a network. This is where different encodings represent characters differently.

FYI There are actually more than 65536 characters in Unicode, so some have to be stored as pairs of *surrogate characters*. Most of the time you don't need to worry about this—most useful characters are in the Basic Multilingual Plane (BMP).

The System.Text.Encoding class is at the heart of .NET's encoding functionality. Various classes are derived from it, but you rarely need to access them directly. Instead, properties of the Encoding class provide instances for various common encodings. Others (such as ones using Windows code pages) are obtained by calling the relevant Encoding constructor. Table 8 describes the encodings you're most likely to come across.

Name	How To Create	Description
UTF-8	Encoding.UTF8	The most common multi-byte representation, where ASCII characters are always represented as single bytes, but other characters can take more—up to 3 bytes for a character within the BMP. This is usually the encoding used by .NET if you don't specify one (for instance, when creating a StreamReader). When in doubt, UTF-8 is a good choice of encoding.
System default	Encoding.Default	This is the default encoding for your operating system—which is not the same as it being the default for .NET APIs! It's typically a Windows code page—1252 is the most common value for Western Europe and the US, for example.
UTF-16	Encoding.Unicode, Encoding.BigEndianUnicode	UTF-16 represents each character in a .NET string as 2 bytes, whatever its value. Encoding.Unicode is little-endian, as opposed to Encoding.BigEndianUnicode.
ASCII	Encoding.ASCII	ASCII contains Unicode values 0-127. It does not include any accented or "special" characters. "Extended ASCII" is an ambiguous term usually used to describe one of the Windows code pages.
Windows code page	Encoding.GetEncoding(page)	If you need a Windows code page encoding other than the default, use Encoding.GetEncoding(Int32).
ISO-8859-1 ISO-Latin-1	Encoding.GetEncoding(28591)	Windows code page 28591 is also known as ISO-Latin-1 or ISO-8859-1, which is reasonably common outside Windows.
UTF-7	Encoding.UTF7	This is almost solely used in email, and you're unlikely to need to use it. I only mention it because many people think they've got UTF7-encoded text when it's actually a different encoding entirely.

Table 8. Common text encodings

→

THREADING

Threading is an exciting field, but it can be hugely complex to implement. In theory, if you're not too ambitious, it should be simple to follow these rules. In practice, keeping track of what's going on can be extremely tricky.

For Windows Forms and WPF:

- Don't perform any tasks on the UI thread which may take a long time or block. This will result in an unresponsive UI.
- Don't access controls from a non-UI thread except as a way to invoke an operation on the UI thread (with Control.Invoke/BeginInvoke for Windows Forms and Dispatcher.Invoke/BeginInvoke for WPF).
- When invoking an operation on the UI thread with Invoke (which blocks until the operation completes) avoid holding any locks when you make the call—if the UI thread attempts to acquire the same lock, your program will deadlock.

In general:

- If the same data is going to be accessed and potentially changed in more than one thread, you'll need to synchronize access, usually with a lock (C#) or SyncLock (VB) statement. Immutable objects can be freely shared between threads.
- Try to avoid locking for any longer than you have to. Be careful what you call while you own a lock—if the code you call acquires any locks as well, you could end up with a deadlock.
- Avoid locking on references which other code may try to lock on—in particular, avoid locking on `this` (C#) / `Me` (VB), and instances of Type or String. A private read-only variable created solely for the purpose of locking is usually a good idea.
- If you ever need to acquire more than one lock at a time, make sure you always acquire those locks in the same order. Deadlock occurs when one thread owns lock A and tries to acquire lock B, while another thread owns lock B and is trying to acquire lock A.
- Suspending, interrupting or aborting a thread can leave your application in a highly unpredictable state unless you are extremely careful. These are inappropriate actions in almost all situations: consult a dedicated threading book before using them.
- When waiting on one thread for something to occur on another, rather than going round a loop and sleeping each time, use Monitor.Wait or WaitHandle.WaitOne, and signal the monitor or handle in the other thread. This is more efficient during the wait and more responsive when you can proceed.

When using .NET 3.5, prefer ReaderWriterLockSlim over ReaderWriterLock—it performs better, has simpler characteristics and provides fewer opportunities for deadlock.

Monitors and wait handles

Monitors are the "native" synchronization primitives in .NET. There is a monitor logically associated with every object (although the monitor is actually lazily created when it's first needed). All members of the System.Threading.Monitor class are static—you pass in an object reference, and it is that object's monitor which is used by the method.

Monitors and wait handles, continued

Method	Description
Enter	Acquires the monitor (used automatically by lock/SyncLock)
Exit	Releases the monitor (used automatically by lock/SyncLock)
TryEnter	Attempts to acquire the monitor, with a timeout
Wait	Releases the monitor (temporarily) and then blocks until it's pulsed
Pulse	Unblocks a single thread waiting on the monitor
PulseAll	Unblocks all threads waiting on the monitor

Table 9. Methods of the System.Threading.Monitor class

Wait handles are .NET wrappers around the Win32 synchronization primitives, and all derive from the System.Threading.WaitHandle class. Wait handles provide some extra abilities over monitors—in particular, as well as being able to wait until one handle is available (WaitOne), you can wait on multiple handles at a time, either until they're *all* available (WaitAll) or until any *one* of them is available (WaitAny). In addition, wait handles can be used across multiple processes for inter-process synchronization. There are four commonly used WaitHandle subclasses, shown in table 10.

WaitHandle Subclass	Description
A Mutex acts quite like a monitor.	A single thread can acquire the mutex multiple times by waiting on it; the same thread has to release the mutex with ReleaseMutex as many times as it has acquired it before any other threads can acquire it. Only the thread which owns the mutex can release it.
A Semaphore has a count associated with it.	The initial value of the semaphore is specified in the constructor call. When a thread waits on a semaphore, if the count is greater than zero it is decreased and the call completes. If the count is zero, the thread blocks until the count is increased by another thread. The Release method increases the count, and can be called from any thread.
AutoResetEvent and ManualResetEvent wait handles both logically have a single piece of state: the event is either signaled or not	An event is signaled with the Set method. In both cases a thread calling one of the Wait methods will block if the event isn't signaled. The difference between the two is that an AutoResetEvent is reset (to the non-signaled state) as soon as a thread has successfully called Wait on it. A ManualResetEvent stays signaled until Reset is called. Physical metaphors can help to remember this behavior. AutoResetEvent is like a ticket barrier: when a ticket has been inserted, the barrier opens but only allows one person to go through it. ManualResetEvent is like a gate in a field: once it's opened, many people can go through it until it is manually closed again.

Table 10. WaitHandle subclasses

More details are available in my threading tutorial at http://pobox.com/~skeet/csharp/threads or consult Joe Duffy's *Concurrent Programming on Windows* (Addison-Wesley Professional, 2008) for a truly deep dive into this fascinating area.

USING C# 3.0 AND VB 9.0 WHEN TARGETING .NET 2.0 AND 3.0

Some C# 3.0 and VB 9.0 features can be used freely when building a project in VS 2008 which targets .NET 2.0; some require a bit of extra work; a couple don't work at all.

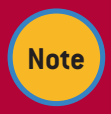
Fully available features

Automatically implemented properties, implicitly typed local variables and arrays, object and collection initializers, anonymous types, partial methods, and lambda expressions (converted to delegate instances) can all be used at will. Note that lamdba expressions are slightly less useful in .NET 2.0 without the Func and Action families of delegate types, but these can easily be declared in your own code to make it ".NET 3.5-ready".

Using C# 3.0 AND VB 9.0 When Targeting .NET 2.0 and 3.0, continued

Partially available features—Extension methods and query expressions

Extension methods require an attribute which is normally part of .NET 3.5. However, you can define it yourself, at which point you can write and use extension methods to your heart's delight. Just cut and paste the declaration for ExtensionAttribute from http://msdn.microsoft.com/library/System.Runtime.CompilerServices.ExtensionAttribute.aspx into your own project, declaring it in the System.Runtime.CompilerServices namespace.



In VB you have to have an empty root namespace for this to work—if necessary, create a separate class library project just to hold this attribute.

Query expressions themselves are available regardless of framework version, as they're only translations into "normal" C# 3.0 and VB 9.0. However, they're not much good unless you've got something to implement, such as Select and Where methods. These are normally part of .NET 3.5, but LINQBridge (http://www.albahari.com/nutshell/linqbridge.html) is an implementation of LINQ to Objects for .NET 2.0. This allows in-process querying with query expressions. (LINQBridge also contains the ExtensionAttribute mentioned earlier.)

Unavailable features—Expression trees and XML literals

As far as I'm aware, there's no way to get the compiler to create expression trees when using .NET 2.0, not least because all the expression tree library classes are part of .NET 3.5. It's just possible that there may be a way to reimplement them just as LINQBridge reimplements LINQ to Objects, but I wouldn't hold your breath—and it would be much more complicated to do this. As a corollary, you can't use "out of process" LINQ without .NET 3.5, as that relies on expression trees.

XML literals in VB 9.0 rely on LINQ to XML, which is part of .NET 3.5, so it's unavailable when targeting .NET 2.0.

ABOUT THE AUTHOR

Jon Skeet

Jon Skeet is a software engineer with experience in both C# and Java, currently working for Google in the UK. Jon has been a C# MVP since 2003, helping the community through his newsgroup posts, popular web articles, and a blog covering C# and Java. Jon's recent book *C# in Depth* looks at the details of C# 2 and 3, providing an ideal guide and reference for those who know C# 1 but want to gain expertise in the newer features.

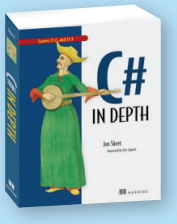
Publications

Author of *C# in Depth* (Manning, 2008), co-author of *Groovy in Action* (Manning, 2007)

Blog
http://msmvps.com/jon.skeet

Web Site
http://pobox.com/~skeet/csharp

RECOMMENDED BOOK



C# in Depth is designed to bring you to a new level of programming skill. It dives deeply into key C# topics—in particular the new ones in C# 2 and 3. Make your code more expressive, readable and powerful than ever with LINQ and a host of supporting features.

BUY NOW
books.dzone.com/books/csharp

Get More FREE Refcardz. Visit refcardz.com now!

Upcoming Refcardz:

Core Seam
Core CSS: Part III
Hibernate Search
Equinox
EMF
XML
JSP Expression Language
ALM Best Practices
HTML and XHTML

Available:

Essential Ruby
Essential MySQL
JUnit and EasyMock
Getting Started with MyEclipse
Spring Annotations
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET
Very First Steps in Flex
C#
Groovy
NetBeans IDE 6.1 Java Editor
RSS and Atom
GlassFish Application Server
Silverlight 2

Visit refcardz.com for a complete listing of available Refcardz.

Design Patterns
Published June 2008

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-16-5
ISBN-10: 1-934238-16-3
50795
9 781934 238165
$7.95